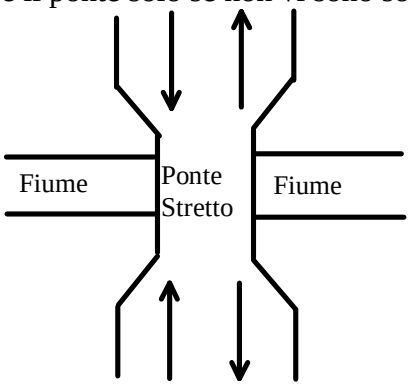
Esercizio di Sincronizzazione Tra Processi: Ponte a Senso Unico Alternato

Un ponte contiene una sola corsia di traffico consentendo così l'accesso a macchine provenienti da una sola direzione per volta, a senso unico alternato.
Si identifichi ciascuna macchina con un processo (ci saranno quindi due tipi di processo, le macchine provenienti da nord e le macchine provenienti da sud) e si scriva un programma che sincronizzi l'accesso delle auto sul ponte, facendo uso dei costrutti monitor e regioni critiche condizionali.
Si tenga conto che:

- non esistono priorità tra i processi;
- un processo può attraversare il ponte solo se non vi sono sopra processi dell'altro tipo.



Soluzioni (in linguaggio pseudo pascal)

Con il Costrutto Monitor

```
progam ponteasensounicoalaternato ;
type dir = ( nord, sud );
```

{I processi macchina potranno distinguersi per tipo semplicemente attraverso l'uso di un parametro che ne indichi la direzione (per esempio, NORD e SUD). Per sincronizzare correttamente l'accesso i processi, prima di accedere al ponte dovranno analizzarne lo stato, verificare l'esistenza delle condizioni di accesso, e, se possono entrare, settare lo stato in modo da notificare la loro presenza sul ponte. Uscendo dal ponte, i processi dovranno modificare lo stato della risorsa e provvedere a sbloccare eventuali processi che si fossero bloccati all'ingresso del ponte. Da ciò consegue la seguente struttura per i processi "auto"}

```
type auto = process ( d:dir )  { processo parametrico per individuare le due possibili direzioni }
begin
    repeat
        ponte.ENTRA( d);
            <transita >
        ponte.ESCI ( d);
    until false;
end;
```

{ Il monitor che servirà per sincronizzare l'ingresso al ponte dovrà contenere tra le sue variabili quelle che servono per descrivere lo stato della risorsa più tutte le variabili condition che servono per poter sospendere e risvegliare correttamente i processi. Nel nostro caso serve mantenere una variabile (dircor) che indichi in che direzione le auto sul ponte stanno attualmente transitando e una variabile intera (nauto) che memorizzi quante sono in totale queste auto. Serve una sola varabile condition per sospendere le auto, poiché non possiamo avere contemporaneamente auto sospese in entrambe le direzioni.}

```
type bridge = monitor ;
var      dircor    : dir;     { verso corrente di percorrenza }
         coda      : condition;  { coda per le auto sospese }
         nauto     : integer;     {numero di auto sul ponte}

procedure entry ENTRA ( d: dir );
begin     if dircor <> d and nauto <> 0  then   coda.wait;
             { se ci sono auto in transito nella direzione opposta è
necessario sospendersi }
         dircor := d; nauto := nauto +1; {una volta entrato setto la
mia presenza sul ponte }
end,

procedure entry ESCI;
{qui non serve il parametro perché dir=dircor }
begin     nauto := nauto -1 ; {setto la mia uscita dal ponte }
         if nauto = 0 then
```

{ Segnalazione di tutte la auto nella direzione opposta. Nota: il processo segnalante è l'ultimo in una certa direzione; anche se segnala tutte le auto nella direzione opposta, e queste sono riattivate immediatamente (soluzione a urgentqueue), eventuali altri processi che arrivano nella direzione del segnalante sono automaticamente esclusi dall'accedere al monitor la cui consistenza è sempre mantenuta; sulla coda possono essere presenti solo auto che rappresentano processi della stessa direzione, diversa da quella corrente }

```
                  while coda.queue do coda.signal ;
end;

begin { inizializzazione del monitor }
    nauto := 0 ; dircor := sud ; { scelta indifferente }
end;

var   ponte : bridge { creazione del monitor }
       auto1, auto2, ..... : auto (nord);
       auto100, auto101, .... : auto(sud);
     ...
```

begin { applicazione parallela con processi e monitor }
end.

{Si consideri la usuale alternativa con segnalazioni a 'catena': il processo segnalante segnala solo il primo sospeso, che a sua volta segnala il secondo, etc. Si confronti l'ordine di acquisizione della risorsa rispetto all'ordine dei processi in coda.}

Con il costrutto regioni critiche condizionali

{Le variabili che tengono conto dello stato del ponte saranno le stesse che con il monitor anche se qui ovviamente non avremo più variabili condition }

```
VAR ponte : shared record
    nauto : integer;  {contatore delle auto che hanno acquisito il ponte }
     dircor  :  dir;        {direzione corrente del ponte}
end record;

procedure ENTRA (miadir : d);   {questa volta sono normalissime procedure }
begin
  {si entra nella regione, prima per prenotarsi, poi per accedere realmente}
region ponte when
        ((miadir = dircor) or (nauto = 0)) do
                dircor := miadir; nauto := nauto + 1;
        end;
end;

procedure ESCI (miadir : d);
begin
region ponte do
        nauto:=nauto-1;  {con le regioni critiche non si deve svegliare nessuno }
end;

type utente = process;
var direzione : dir;
begin
   loop   <scegli dir>
                ENTRA (direzione);
                <passa sul ponte>
                ESCI (direzione);
        end loop;
end utente;

var u1, ..., un : utente;

begin
        region ponte do
                nauto := 0; dircor :=su;
end;
```

Con l'uso di ADA

{ Per risolvere il tutto con ADA possiamo costruire un processo che serva come gestore.
Notiamo che la entry ACQUIS per il processo ponte deve essere differenziata per direzione: infatti, il processo servitore puo' conoscere le richieste solo servendole, e quindi non avrebbe piu' modo di ritardare le richieste accettate ma non corrette in direzione.
Quindi si devono distinguere le entry in modo che il servitore sappia quali (e quando) servire le richieste in arrivo.}

```
package body TuttoIlPonte is

type dir : ( nord, sud ),

task typeautosu is
end autosu;

task body autosu is
begin
        loop
                ponte.ENTRAsu;
                <transita >
        ponte.ESCI;
        end loop;
end autosu;

task type autogiu is
end autogiu;

task body autogiu is
        begin
        loop
                ponte.ENTRAgiu;
                        <transita >
                        ponte.ESCI;
                end loop;
end autogiu;

task ponte is
  entry ENTRAsu;
  entry ENTRAgiu;
  entry ESCI;
end ponte;

task body ponte is
        dircor    : dir; { verso corrente di percorrenza }
        nauto     : integer;
begin  nauto := 0; dircor := su;
```

```
loop
 select
   when (dircor  = su) and (nauto <> 0) or (nauto = 0) ==>
   accept ACQUISsu  do
        nauto + := 1;
        if dircor <> su then dircor := su endif;
     end ACQUISsu;
 or
   when (dircor  = giu) and (nauto <> 0) or (nauto = 0) ==>
   accept ACQUISgiu  do
        nauto + := 1;
        if dircor <> giu then dircor := giu endif;
     end ACQUISsu;
 or
   accept RILAS do nauto - := 1;
     end RILAS;
 end select;
end loop;
end ponte;
 asu1, asu22, ...  : autosu;
 agiu1, agiu2, ... : autogiu;

end TuttoIlPonte;
```

{ Notiamo che non c'e' nessuna garanzia che non si ottenga **starvation** per i processi in una direzione: infatti, il costrutto select è nondeterministico. Quindi a fronte di traffico in entrambi i sensi è possibile una scelta 'sempre' a favore della stessa direzione.
Se vogliamo tenere conto di questo si puo' introdurre una alternanza stretta dei versi di percorrenza. Si scriva la **soluzione a alternanza stretta**.
Notiamo che una stretta alternanza puo' portare a non utilizzare al meglio la risorsa.
La seguente soluzione usa l'alternanza in caso di traffico in entrambe le direzioni e fornisce accesso ad una qualunque direzione in caso che l'altra si priva di richieste.
(si esamina solo il task gestore, la parte di interfaccia presenta una nuova entry di richiesta, che precede l'acquisizione) }

```
task ponte is
  entry Richiesta (d: in dir);
  entry ACQUISsu;
  entry ACQUISgiu;
  entry RILAS;
end ponte;

task body ponte is
        dircor    : dir; { verso corrente di percorrenza }
        nauto     : integer;          turno     : dir; { verso di alternanza corrente }
        ric       : array [dir] of integer;

begin  nauto := 0; dircor := su; turno := su;
```

```
loop
 select
   accept Richiesta (d: in dir) do ric [d] + := 1 end Richiesta;
 or
   when (dircor  = su) and (nauto <> 0) or
        ((nauto = 0) and (ric [giu]= 0) or
        ((nauto = 0) and (turno = su)      )  ==>
   accept ACQUISsu  do
        nauto + := 1; ric [su] - := 1;
        if dircor <> su then dircor := su endif;
     end ACQUISsu;
 or
   when (dircor  = giu) and (nauto <> 0) or
        ((nauto = 0) and (ric [su]= 0) or
        ((nauto = 0) and (turno = giu)     )  ==>
   accept ACQUISgiu  do
        nauto + := 1;  ric [su] - := 1;
        if dircor <> giu then dircor := giu endif;
     end ACQUISsu;
 or
   accept RILAS do nauto - := 1;
                  if nauto = 0 then turno := altro (turno);
     end RILAS;
 end select;
end loop;
end ponte;
```

{Si accettano richieste in una direzione determinata:

- se ci sono **gia' servizi** in corso nella stessa;
- se **non ci sono richieste** nella direzione opposta;
- in caso di traffico (richieste) in entrambe, la variabile **turno** stabilisce la direzione di servizio; la variabile turno viene gestita secondo una politica di stretta alternanza. }

Con l'uso di JAVA

/*
In Java si possono definire oggetti "Thread" che sono oggetti che hanno associato un metodo run che, invocato automaticamente dopo lo start() dell'oggetto, rende l'oggetto un flusso di esecuzione indipendente.
Due o più Thread che eseguono in parallelo possono fare richieste di servizio concorrentemente ad un oggetto di cui mantengono entrambi il riferimento. In Java è possibile usare tali oggetti come se fossero dei monitor. Con alcune differenze e limitazioni.
I metodi di un oggetto che può venire acceduto concorrentemente da più Thread possono essere resi mutuamente esclusivi nella loro esecuzione (esattamente come quelli di un monitor) attraverso la direttiva synchronized.
Ad ogni oggetto che abbia dei metodi synchronzed è associata una e una sola varaibile condition (che, essendo unica non ha nemmeno bisogno di un nome di riferimento simbolico). Su tale variabile condition sono possibili tre operazioni:
wait(); sospende in coda il processo che la invoca come per le normali condition
notify(); risveglia uno dei processi in coda; non si sa quale in verità
notifyAll(); risveglia tutti i processi in coda; non ci dice quale acquisirà per primo il monitor

In questo esercizio il fatto di avere una sola variabile condition a disposizione non crea problemi perché effettivamente ce ne serve solo una. Quindi lo risolviamo esattamente come lo avevamo risolto il monitor

Notiamo che questa volta non facciamo uso di pseudo-codice ma bensì di vero e proprio codice Java che è effettivamente compilabile e eseguibile.
*/

```
// questo è il main del programma che crea l'oggetto monitor e lancia gli oggetti thread
// la direzione è indicata dal valore, 1 o 2, di una variabile intera
class PonteSemplice {
    public static void main (String args[]) {
    PonteMonitor ponte = new PonteMonitor();
    for (int i = 0; i<5 ; i++) {
        new Macchina(("Nord"+i),ponte,1).start();
        new Macchina(("Sud"+i),ponte,2).start();
        }
  }
}

// questo è il codice dell'oggetto monitor
class PonteMonitor {
    int nauto;
    int dircor;
    public PonteMonitor() { nauto=0; dircor=1;}
    public synchronized void Entra(int dir) {
        if ( nauto != 0 && dircor != dir) {
            try {wait();} catch (InterruptedException e) {} }
// in Java tutte le chiamate sospensive possono essere soggette ad una eccezione che
// deve essere catturata, e quindi vanno inserite all'interno di un blocco di cattura
// esso è il try {chiamata bloccante} catch (..) {}
        nauto ++;
        dircor = dir;
```

```
        System.out.println("Entro nel ponte");
        }
    public synchronized void Esci() {
        nauto--;
        if(nauto==0) notifyAll();
        System.out.println("Esco dal ponte");
        }
}
```

// questo è il codice dell'oggetto derivato dalla classe thread che realizza il processo macchina

```
class Macchina extends Thread {
   PonteMonitor ponte;
   int miadir;

   public Macchina (String str, PonteMonitor ponte, int miaidir) {
    super(str);
    this.ponte = ponte;
    this.miadir = miadir;
}
    public void run() {

        for(;;) {
                ponte.Entra(miadir);
                try {sleep(1000);} catch (InterruptedException e) {}
                ponte.Esci();
                    }
            }
}
```